Technical Debt Reduction using Search Based Automated Refactoring

Technical Debt Reduction using Search Based Automated Refactoring

Michael Mohan*, Des Greer, Paul McMullan
Department of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast
BT7 1NN, Northern Ireland, UK

Abstract

Software refactoring has been recognised as a valuable process during software development and is often aimed at repaying technical debt. Technical debt arises when a software product has been built or amended without full care for structure and extensibility. Refactoring is useful to keep technical debt low and if it can be automated there are obvious efficiency benefits. Using a combination of automated refactoring techniques, software metrics and metaheuristic searches, an automated refactoring tool can improve the structure of a software system without affecting its functionality. In this paper, four different refactoring approaches are compared using an automated software refactoring tool. Weighted sums of metrics are used to form different fitness functions that drive the search process towards certain aspects of software quality. Metrics are combined to measure coupling, abstraction and inheritance and a fourth fitness function is proposed to measure reduction in technical debt. The 4 functions are compared against each other using 3 different searches on 6 different open source programs. Four out of the 6 programs show a larger improvement in the technical debt function after the search based refactoring process. The results show that the technical debt function is useful for assessing improvement in quality.

Keywords: search based software engineering, automated refactoring, refactoring tools, technical debt, software metrics, simulated annealing

1. Introduction

Search based software engineering is an area that tries to apply search heuristics to solve complex problems in software development. It has been used to help resolve problems in software design, project management, software release planning, model verification and software testing (Harman et al., 2012a). Search based techniques can be used to provide automated assistance in areas of software management to save resources on a development project.

The term 'Technical Debt' (TD) refers to a metaphor and has been defined as "the trading of long-term software quality in favour of short-term expediency" (Brown et al., 2010). In other words TD occurs where long-term software quality, and therefore ease of maintainability, is temporarily sacrificed with the expectation that it will be improved in the near future. The sacrifice may be in terms of design and could be due to not having enough knowledge of the problem being solved or just an urgent need to make and demonstrate progress. In any case, debt accumulates interest and it becomes more expensive to repay with time. With time it becomes harder to add functionality due to structural issues becoming more critical and the occurrence of defects becomes more likely. To improve the long term efficiency of a project and to lower its operational risk, the TD can be kept to a minimum by making regular repayments, meaning refactorings. The negative side of this is that time spent on refactoring will in turn decrease the amount of time used to add functionality to software. Therefore, any approach that makes this easier or even automatic is likely to be financially beneficial.

Search Based Software Maintenance (SBSM) uses search based software algorithms to tackle this problem. By applying automated refactoring techniques that modify the structure of a software program without affecting the functionality, this process can be applied without the direct involvement of the programmer, allowing time to concentrate on other aspects of the project. SBSM treats the maintenance of a software system as a combinatorial optimisation problem. The software code represents the search space of the problem and the refactorings can be applied across this search space to explore possible solutions. As there would be too many possible changes to software program to permit an exhaustive search of the software

* Corresponding author.
E-mail address: mmohan03@qub.ac.uk (M. Mohan).

space, metaheuristic search techniques can be used to seek out the most optimal solutions.

The search techniques can analyse a software program using some measure of quality to improve the structure or decrease the TD in the program. Using a set of software metrics, the search can then work towards an optimal solution in a more realistic time frame. In this paper it is shown how automated refactoring techniques, metaheuristic search approaches and software metrics can be used together to reduce TD. The automated aspect of the process allows a search to converge towards an optimal state over numerous iterations. It also allows the programmer to focus on other issues, freeing up a large proportion of time and reducing maintenance effort.

An issue present with this approach is that software development is not a straightforward process. There is a lot of uncertainty involved. In order to increase the maintainability of the software there needs to be a measure with which to compare. However software quality is not easy to measure. There can be various different properties to balance in the structure of a program and there may be conflicting interests. Furthermore, depending on certain factors (such as the type of program being developed or the programming language used), the important aspects of a program may be different. With object oriented programming, considerable effort has been made to establish important properties in a well-structured program (Martin, 2000). Metrics have been introduced to measure aspects of program structure and behaviour, but finding a balance between the different aspects can become difficult when there are contradictions between them. Likewise, it may be uncertain which aspects of a software program design should be prioritised.

The goal of this paper is to investigate the effectiveness of using TD to direct automatic refactoring. We wish to know if TD can be used effectively as a fitness function for search based automatic refactoring. To address this, using Basili's Goal Question Metric approach (Basili et al., 1994), we derive the following question:

RQ1: How does a fitness function for Technical Debt compare with some other commonly used design quality metrics?

To consider this we can look at deriving a TD metric and comparing it against metrics based on levels of *abstraction*, *coupling* and *inheritance*, all of which are well established as design quality factors (Bansiya and Davis, 2002). These properties have been chosen to represent individual quality indicators as they can represent a range of different aspects of software measurement. Table 1 gives a short description of each property and how it is calculated. The details of the calculations used to represent each property are given in Table 6. Inheritance will be a good indication of whether the design is inefficient and whether the classes are related and extended properly. Inheritance is concerned with measuring how the objects in a project are organised hierarchically, so class level metrics are used to represent it. The measure incorporates interface implementation and use of abstract classes, and so a high measure is considered desirable. Coupling can be used to derive how the extent to which the objects in a software system depend on each other, generally expected to be as low as possible. Abstraction will indicate the amount of changes needed between specific objects in order to implement new additions to the system. Again, a high value here is considered better. As previous work in the area has investigated abstraction (O'Keeffe and Ó Cinnéide, 2003), (Mitchell and Mancoridis, 2002), coupling (Veerappa and Harrison, 2013), (Murgia et al., 2012) and inheritance (O'Keeffe and Ó Cinnéide, 2007), there is some support for the position that these are useful properties to use for a comparative study against an approach for tackling TD.

Table 1. Individual Quality Properties

Property	**Context**
Abstraction	How easy it is for a software system to be extended and built upon. Estimated based on number of abstract classes present and the number of interfaces present and implemented.
Coupling	A measure of the dependencies between classes based on counts of usage of class, attributes and parameters by other classes.
Inheritance	A measure of the class structure of a project in terms of counts of interface implementations and of descendants and ancestors.

To further the investigation, an experiment has been conducted using the refactoring tool A-CMA (Koc et al., 2012) to assess the effectiveness of three sets of metrics that measure these object oriented properties and compare them against a proposed set of metrics to measure TD. A weighted sum is used to combine the metrics into an overall score to improve. Thus the following hypothesis and null hypothesis are to be tested:

H1: Technical Debt can be reduced significantly using search based automatic refactoring.

H1_0: There is no effect on Technical Debt after search based automatic refactoring.

A further question investigated is:

RQ2: How does a simulated annealing search perform compared to hill climbing and a random search in a search based automated refactoring approach to address Technical Debt?

Again the same metrics can be used i.e. TD reduction, abstraction gain, coupling reduction, inheritance gain but also execution time. From this we can postulate as follows:

H2: Simulated annealing performs better than hill climbing/random search for search based automatic refactoring to reduce Technical Debt.

H2_0: There is no significant difference in the effectiveness between simulated annealing and hill climbing/random search for search based automatic refactoring to reduce Technical Debt.

The remainder of the paper will be structured as follows. Section 2 details the automated refactoring tool used and the different components available in the tool. Section 3 outlines the experiment conducted and the metric functions measured. Section 4 then details the outcome of the experiment and analysis of the results. Section 5 identifies threats to validity in the experiment. Section 6 outlines related work in the area of SBSM. Finally, conclusions are made in Section 7 and possible directions for future work are discussed in Section 8.

2. Refactoring Tool

A-CMA is an automated refactoring tool developed by Koc et al. (2012) that refactors Java programs using Java bytecode as input. An advantage of this tool over many others is that it has many options for refactoring as well as metrics available and it is highly configurable. The tool has the option to create and select different configurations of metrics and refactoring actions. This can be selected on the application and allows different metrics and actions to be enabled. It also gives the option to apply different weights to the metrics. This allows the user to construct different metric combinations that can be used on a task. An overall metric score is derived using a weighted sum of each enabled metric. A popular approach in recent literature is to use a multi-objective pareto approach to derive an overall metric score although this approach is non-deterministic. A weighted sum allows for some metrics to be given more influence than others, in order to reflect their importance. The metrics have the ability to be specified as maximized or minimized. Maximized metrics are metrics where an increase in value causes an improvement and minimized metrics are metrics where a decrease in value causes an improvement. The overall quality gain of a task can be derived by finding out how much the overall score has reduced. Metric details are loaded in from an xml file and maximized/minimized metrics can be specified here.

Before the experiment was conducted, some changes were made to the existing tool for the purposes of this paper[1]. Extra initial parameters were included for the hill climbing and simulated annealing searches, in order to allow more control when configuring a search task (the option was given to input a starting temperature for simulated annealing and to indicate first ascent or steepest ascent hill climbing). Increased control was given over the configuration of tasks and improved data output was configured including quality gain and average values. This allowed for the tasks to be loaded into the program and run one after the other with all available parameters configured and all available data captured. The ability to incorporate maximized (those for whom an increase is desirable) metrics was also implemented, allowing maximized and minimized (those for whom a decrease is desirable) metrics to be combined into an overall weighted sum. Finally, more control was given to the configurations used to create different fitness functions, with the ability to enable/disable specific metrics.

2.1 Metaheuristic Search

The tool has the ability to run 5 different searches with 10 different variations but for the purposes of this paper only 3 are used. Initially a random search is run to provide a benchmark against which the other searches can compare. A random search simply applies refactoring actions at random and measures the score after each iteration. The only input option available for this search is to specify the number of iterations needed. After the specified number have completed, the best score is taken as the final result. The 2 heuristic searches, hill climbing and simulated annealing, were chosen as they are used commonly in the research and therefore can be compared against other work in the area e.g. (O'Keeffe and Ó Cinnéide, 2008), and because they are relatively easy to implement and modify for the purposes of the experiment.

The first of the 2 heuristic searches used is a hill climbing algorithm. This is a local search that finds

[1] The original tool can be found at https://github.com/eknkc/a-cma and the updated version at https://github.com/mmohan01/a-cma

a local optimum solution by comparing neighbouring changes in the solution space (Räihä, 2009). Numerous variations of this search can be chosen in the tool. Firstly, one can choose to either select first ascent hill climbing or steepest ascent. First ascent will find the first neighbour with an improved score and use it for the next iteration. Steepest ascent compares each available neighbour to find the option with the greatest improvement. This can result in a better search and more optimum values found but can take longer than first ascent. Also, the search can be selected as a multiple starting algorithm or a single start. A multiple start hill climbing algorithm will begin the search again at a different point in the solution space after an optimum solution is found, giving the possibility to find a better optimum at a different point in the program. The amount of restarts can be specified as well as the depth away from the current solution at which the next starting point is to be found. The A-CMA tool also gives the ability to specify the maximum amount of iterations, at which point the search will terminate if it has not already found the optimum solution.

The other search used was simulated annealing. This is similar in practice to hill climbing, although it allows the ability to accept a solution of worse quality in order to escape local optima. Like hill climbing, it will begin at a random point in the solution space, and apply a refactoring to the solution. The difference is that, when the score for the new solution is calculated, a worse solution may be kept. This is determined by the start "temperature" of the search. The search is named due to being a simulation of the cooling process in metallurgical annealing. The particles in the metal will begin at a high temperature and move about rapidly, inspecting different states. As the temperature of the metal cools, the particles will begin to settle after exploring the different energy states. This allows the metal to become stronger when it finally cools to a solid. Likewise, the simulated annealing search gives the freedom to "explore" different options in the solution space early on in the search, even accepting a certain probability of worse solutions. As the temperature cools, this probability gradually decreases until the solution only accepts better neighbours, essentially becoming identical to the hill climbing algorithm. The value of the starting temperature will determine how rapidly the search "cools" and thus how much freedom the search will have to accept worse quality solutions. With a higher initial temperature the probability will be higher but will drop more rapidly. 2 initial parameters can be set for the annealing algorithm, the starting temperature and amount of iterations in the search.

2.2 Refactoring Actions

Table 2. Field Level Refactorings

Increase Field Security	Increases the security level of a field by one level (between private, package, protected and public)
Decrease Field Security	Decreases the security level of a field by one level
Move Down Field	Moves a field from the current class to a sub class
Move Up Field	Moves a field from the current class to its immediate super class
Remove Field	Removes a field from the class

Table 3. Method Level Refactorings

Increase Method Security	Increases the security level of a method by one level
Decrease Method Security	Decreases the security level of a method by one level
Move Down Method	Moves a method from the current class to a sub class
Move Up Method	Moves a method from the current class to its immediate super class
Move Method	Moves a method from the current class to one of its parameter types
Instantiate Method	Moves a static method from the current class to one of its parameter types
Freeze Method	Sets a method as static
Remove Method	Removes an unused method from the class
Inline Method	Sets the body of a method inside the caller (as long as there is only one caller) and removes the method

Table 4. Class Level Refactorings

Introduce Factory	Creates a new factory method for a class constructor and replaces any references to the constructor with calls to the new method, implementing the factory method design pattern
Make Class Abstract	Makes a class into an abstract class (as long as it hasn't been instantiated elsewhere in the program)
Make Class Final	Makes a non-final class final
Make Class Non-Final	Makes a final class non-final
Remove Class	Removes an empty class from the program
Remove Interface	Removes an empty interface from the program

The A-CMA tool contains 20 available refactoring options to apply on the field, method and class level of a Java program. To apply these refactorings automatically, the available objects are found for each refactoring by excluding objects that are ineligible (for example if a class is already abstract the "Make Class Abstract" refactoring won't be applicable). Once the initial list of available objects has been acquired for each refactoring, they can be chosen and applied stochastically in order with the search algorithm used. The available refactorings are listed and described in Tables 2-4. Many of these refactorings implement refactoring options proposed by Fowler in his book (Fowler, 2002) and on his website (Fowler, 2015).

2.3 Software Metrics

There are 24 metrics available in the A-CMA tool but in the scope of this paper only 17 are used. The metrics used along with description for each one are given in Table 5.

Table 5. Software Metrics Used in Experiment

Identifier	Description
numField	The amount of fields per class
numOps	Number of methods per class
numCls	Number of classes in a package
numInterf	Number of interfaces in a package
iFImpl	Number of interfaces implemented by a class
abstractness	The ratio of abstract class to classes in a package
avrgField Visibility	Average amount of field visibility per class (where field visibility is represented by Private:0, Package:1, Protected:2, Public:3)
nesting	The nesting level per class
NOC	Number of children per class
numDesc	Number of descendants per class
numAnc	Number of ancestors per class
iC_Attr	Number of attributes in a class using another class or interfaces as type
eC_Attr	Number of external uses of a class as attribute type
iC_Par	Number of parameters in class methods using another class or interface as type
eC_Par	Number of external uses of class as parameter type in method
Dep_In	Number of elements that depend on a class
Dep_Out	Number of elements depended on by a class

3. Experimental Design

The experiment aims to compare four different fitness functions that each uses a combination of available metrics to represent some measureable property of software design. In order to compare these fitness functions, each function is given a set of weights for each metric that must add to 1 overall. This way the amount of metrics used in each function will not interfere and the functions will be normalized for comparison against each other. In order to create an overall score from the fitness functions, the direction of improvement of each software metric must be taken into consideration (whether increase in the value causes an improvement or a decrease in the value causes an improvement). Of the 17 metrics used, 10 have been determined to be minimized metrics and the other 7 have been determined to be maximized metrics. The positive/negative aspect of the metrics did not need to be taken into consideration when aggregating the weights to 1.

The goal is to minimise the value of the metric function being inspected in order to improve the symptoms of the property being represented. The evaluation function is given in equation 1, where there are n that make up the fitness function, w_m is the weight of the metric and v_m is the value of the metric. The value d is a binary constant that represents effect of the metric, where an increase is signified by -1 and a decrease is signified by 1.

$$\text{Minimize} \sum_{m=0}^{n} d[w_m v_m] \qquad (1)$$

The weights of all the metrics in the function must add to 1 as shown in equation 2:

$$\sum_{m=0}^{n} w_m = 1 \qquad (2)$$

Three fitness functions were created from the metrics to represent important quality properties of object oriented programs (abstraction, coupling and inheritance), and then a fourth was created to represent TD in the system. In order to choose the relevant metrics and the relative weights to represent the TD score, the SOLID principles of object oriented design (Martin, 2000), as well as the QMOOD metric suite of Bansiya and Davis (2002) were used as a basis in which to represent bad software construction. All available refactoring actions were enabled for the 4 fitness functions to give the maximum potential for change. Table 6 gives details about each fitness function compared along with weights used and whether the metric was maximized or minimized (denoted by '+'/'-').

Table 6. Metric Details for Each Fitness Function (see Table 5 for Metric Descriptions)

Software Property	Metric Components and Weights
Technical Debt	-0.1*numFields - 0.1*avrgFieldVisibility - 0.1*numOps - 0.06*nesting + 0.1*abstractness + 0.1*numCls + 0.1*numInterf + 0.1*iFImpl + 0.06*NOC + 0.06*numDesc - 0.06*Dep_In - 0.06*Dep_Out
Coupling	-0.125*iC_Attr - 0.125*eC_Attr - 0.125*iC_Par - 0.125*eC_Par - 0.25*Dep_In - 0.25*Dep_Out
Inheritance	0.25*iFImpl + 0.25*NOC + 0.25*numDesc + 0.25*numAnc
Abstraction	0.33*abstractness + 0.33*numInterf + 0.33*iFImpl

Of the available software metrics, the most applicable were chosen to represent components of the 3 software properties. Metrics were already grouped together as coupling and inheritance metrics in the A-CMA tool, so these were the metrics used to represent the coupling and inheritance properties. The abstraction property was made up of the three metrics determined to be related to abstraction due to them measuring properties of interfaces present in the software. In most cases, the weights were kept level between the metrics used in each fitness function. For the coupling function, the Dep_In and Dep_Out metrics were given priority over the others as they contained aspects of the other coupling metrics used as part of their calculations.

Table 7. Java Programs Used In Experiment

Name	Classes	KLOC (Approx.)	Initial Refactorings Available
JSON	8	2	167
JFlex	78	9	1094
Apache-XmlRpc	89	4	712
Mango	91	3	598
Beaver	95	6	801
JHotDraw	240	18	3297

For the TD function, the 12 metrics intuitively considered to be most relevant were chosen. Initially the metrics were prioritised into 4 different groups. In order to normalise the weights and allow the metrics to accumulate to 1, these were reduced to 2 different weights; 0.06 to represent the bottom 2 categories and 0.1 to represent the top 2. The nesting, NOC and numDesc metrics were given less priority due to their more descriptive nature compared to the other metrics. In a software system, more nesting more descendants and less classes in a package may not particularly be a bad thing, whereas less classes overall may result in classes with too many responsibilities and the appearance of more code smells. The Dep_In/Dep_Out metrics were deemed less importance as, while dependencies should be minimised between classes, they may be required in certain cases. In all cases metrics and weights chosen were speculative and based on intuition. In some cases directions of improvement also had to be chosen.

Table 8. Java Program Execution Times

Name	Time Taken
JSON	0h 3m 13s
JFlex	2h 6m 38s
Apache-XmlRpc	1h 23m 43s
Mango	1h 1m 29s
Beaver	1h 25m 4s
JHotDraw	49h 28m 4s
Total	55h 28m 11s

Each fitness function was compared using 3 different searches. The random search was used as a benchmark with 5,000 iterations. Steepest ascent hill climbing was chosen for the experiment with 30 restarts at a depth of 5 neighbours (chosen based on published comparisons between different hill climbing parameters (Koc et al., 2012)). The third search used was low temperature simulated annealing (as low temperatures have been found to be more effective by O'Keeffe and Ó Cinnéide (2008)) with 5,000 iterations and with the starting temperature set to 1.5. Each search was conducted 10 times using the 4 fitness functions with average values calculated. The input programs for the experiment consisted of 6 open source Java projects: *JSON*, a Java library for data exchange format; *JFlex*, a lexical analyzer generator; *Apache-XmlRpc*, an XML-based remote procedure call library; *Mango*, a collections library; *Beaver*, a parser generator and *JHotDraw*, a GUI framework for drawing editors. These programs were chosen as they have all been used in previous SBSM studies and so there is an increased ability to compare the results and also because they promote different software structures. Details about the programs are given in Table 7. The total number of runs of the experiment came to 10*3(searches)*4(functions)* 6(benchmarks) for a total of 720 runs. The experiment was carried out on a PC with a 3.40GHz Intel Core i7-3770 processor and 8GB of RAM.

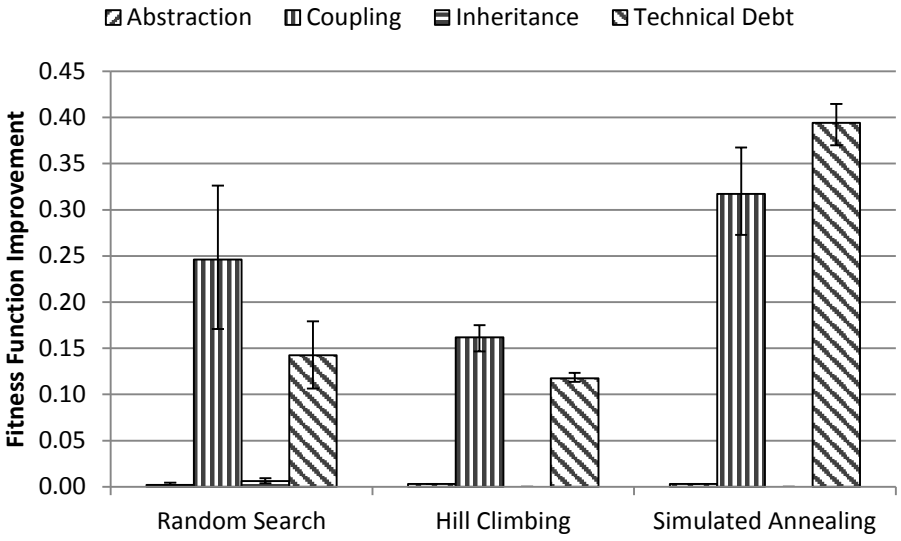

Figure 1. Overall Mean Quality Gain for Each Fitness Function per Search Type

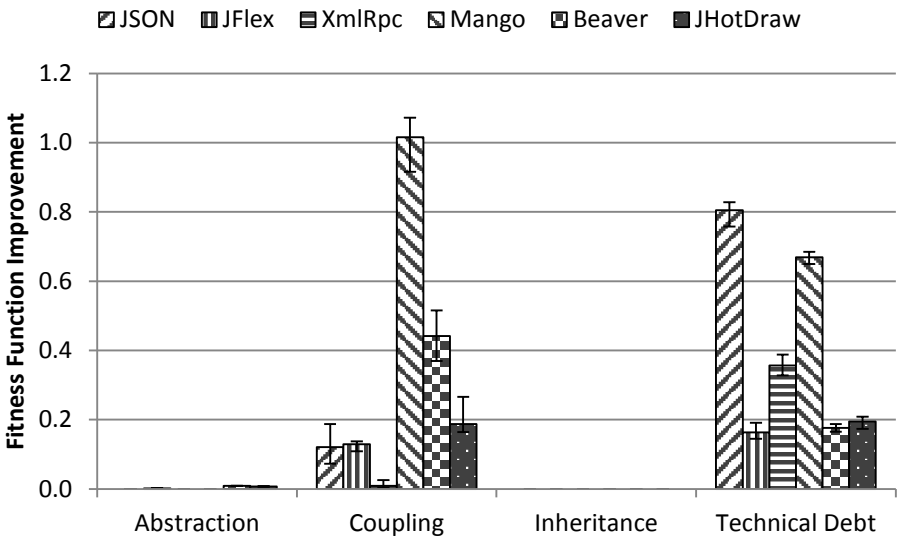

Figure 2. Mean Quality Gain of Each Fitness Function using Simulated Annealing

4 Results

The time taken to complete the tasks for each program is given in Table 8. Clearly here the JHotDraw program caused a bottleneck in execution time and this is most likely due to its size compared to the other projects (containing more than double the amount of classes than the other projects). For instance, JHotDraw contains roughly 18,000 lines of code compared against roughly 9,000 for JFlex, the program with the next longest execution time. It is reasonable to assume that as the project increases, the search space for the refactoring process will increase also giving a large upswing in time taken even with the metaheuristic searches available. This can lead to an increase in time of order n^2. Likewise an attempt to execute the experiment on another open source Java program resulted in 47 hours 40 minutes and 13 seconds taken to run only 4 of the 12 tasks. It contained 408 classes, which seems to support this explanation. These large execution times for certain tasks suggest that a more efficient method is needed to refactor larger programs.

Figure 1 shows the average quality gain across the 6 programs for each fitness function using each of the 3 searches. The results show that simulated annealing gives the highest relative quality improvement, but they also show that the random search outperforms hill climbing.

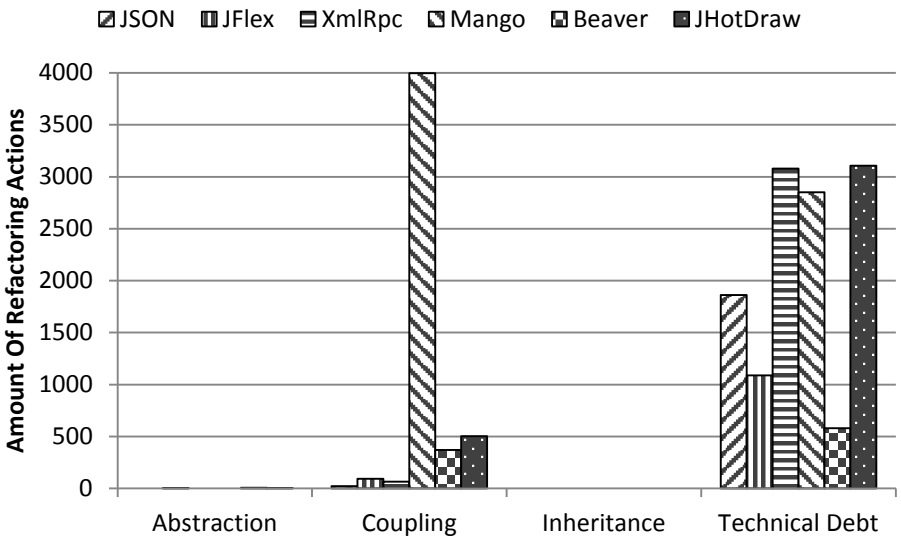

Figure 3. Mean Amount of Actions Applied to Each Fitness Function using Simulated Annealing

The TD quality gain values for each pair of searches were compared using a two-tailed Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level (α = 5%).

The simulated annealing results were analysed to be statistically different when compared against the random search and the hill climbing search across every TD result. The random search results were also found to be significantly different to the hill climbing search. The random search understandably has a larger range of values but the better outcome it gives implies that the hill climbing search was inefficient for the set of tasks. Perhaps the input parameters were not optimal for that search. The simulated annealing and hill climbing searches failed to create any quality gain using the inheritance function whereas the random search yielded a small increase in quality. It is assumed this is due to the freedom and volatility of the random search to find different solutions, but not necessarily to find optimal solutions.

Figure 2 inspects the simulated annealing results, showing the average quality gain for each of the fitness functions across each of the 6 benchmark programs (this is the final overall metric score minus the initial score, averaged over the 10 runs). Of the three individual property fitness functions, coupling seems to be the only one that had shown any significant improvement. The abstraction tasks show minimal improvement and the inheritance tasks had no change at all. In fact, the only case where the inheritance function had any change was in the random search as shown in Figure 1. The TD function was more effective in showing an improvement. The initial and final metric scores for the TD function were statistically analysed using a two-tailed Wilcoxon signed-rank test (for paired data sets) with a 95% confidence level (α = 5%). The obtained results were statistically significant when comparing every run of the TD function. The lack of improvement in the abstraction and inheritance functions implies that there is a lack of volatility in the metrics used to compose these functions.

Figure 3 shows the average amount of applied actions for each of the simulated annealing tasks. These results show a similar trend to the quality gain results and the abstraction and inheritance tasks are similarly devoid of applied refactoring actions. This implies that the reason for the poor quality gain results for those functions stems from the lack of available actions, whereas the other metrics are more volatile and that there are more refactoring actions available to improve them.

Figure 4 gives the overall average applied actions for each fitness function. This continues to show a relationship between the amount of actions available for each fitness function and the quality gain values for the function shown in Figure 2.

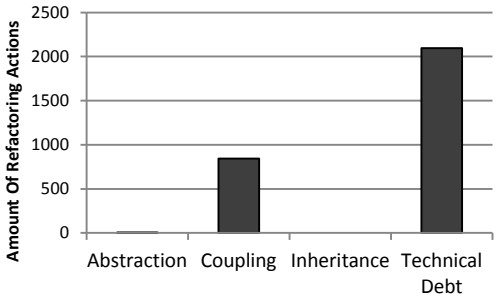

Figure 4. Overall Mean Applied Actions using Simulated Annealing

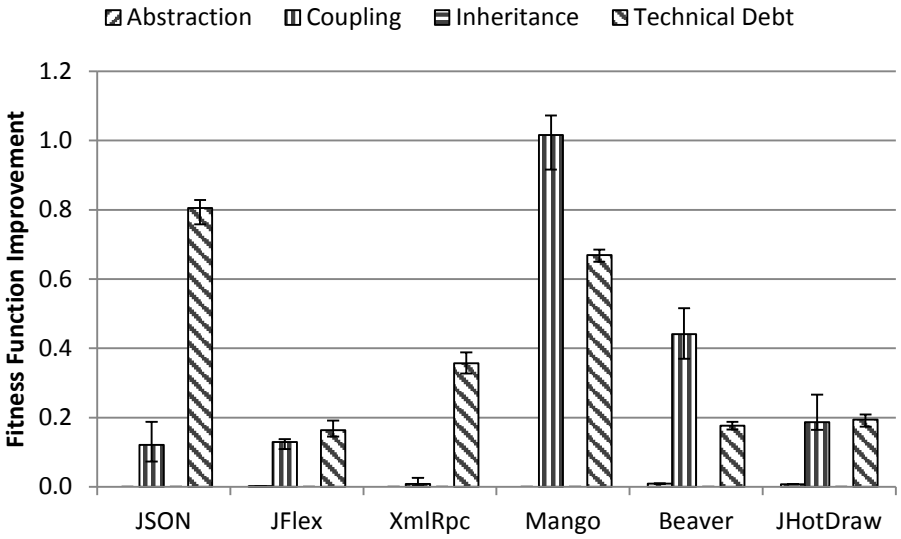

Figure 5. Mean Quality Gain of Each Program using Simulated Annealing

It seems that the volatility of the metrics that make up each function is important to allowing the program to be refactored in any way. The harder the metrics are to improve, the less chance the program will be refactored.

Figure 5 gives another view of the quality gain results, this time highlighting the results for each individual program and allowing a better comparison of the coupling and TD values. Most of the results favour the TD function over the others, but in 2 cases, Mango and Beaver, the coupling function shows higher gains than the TD function by a significant amount. This could suggest that for these 2 programs coupling was high and so amenable to improvement therefore contributing less to the TD calculation. The 2 programs that show the most significant improvement of the TD function over the coupling function are JSON and Apache-XmlRpc. JSON is the smallest program used so perhaps the minimal amount of classes make it harder to reduce the coupling between them as there is minimal coupling in the first place.

Likewise, Apache-XmlRpc contains almost no improvement in coupling implying it too contains little coupling between the classes. The largest quality gain among all the programs was in Mango.

Figure 6 gives the overall average quality gain for each fitness function. It confirms that the TD function had a more significant improvement among the programs than the other 3 fitness functions that represented specific properties. Figure 3 also shows that the TD function involved significantly more refactorings than the other 3 functions.

Figures 7 and 8 show the average quality gain values for each individual metric in the TD and coupling fitness functions (across all 6 benchmarks), giving an idea of the volatility of each metric and their influence on the overall metric scores. In the TD function, only 5 of the 12 metrics show significant quality gain values with the most influential being the numOps metric.

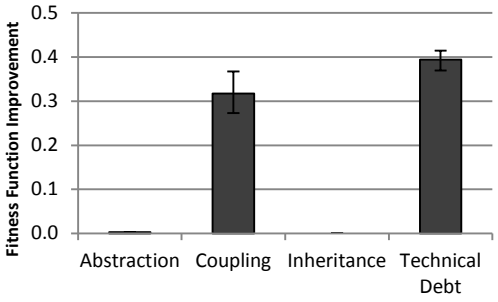

Figure 6. Overall Mean Quality Gain for Each Fitness Function using Simulated Annealing

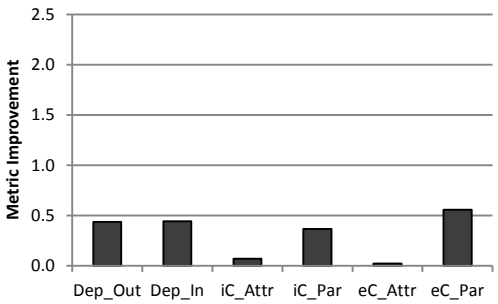

Figure 7. Mean Quality Gain for Each Metric of the Coupling Function using Simulated Annealing

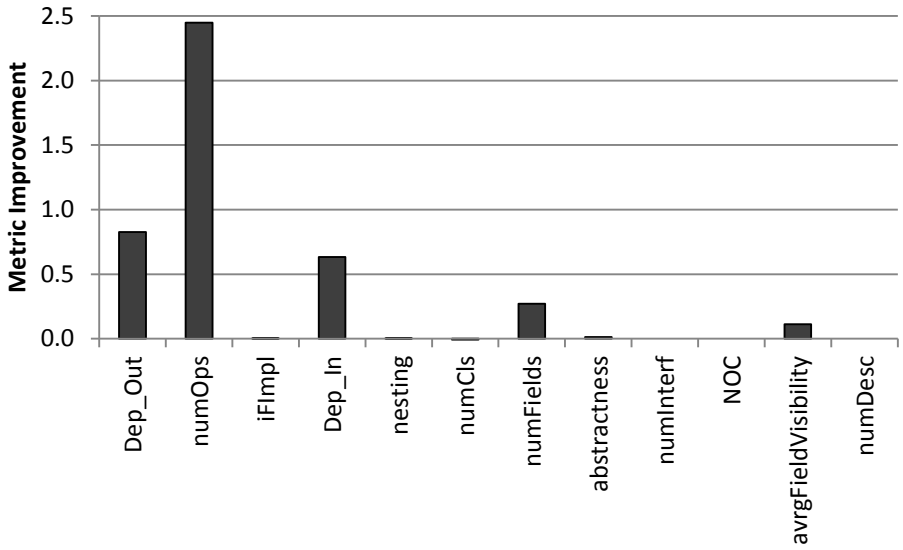

Figure 8. Mean Quality Gain for Each Metric of the TD Function using Simulated Annealing

The numInterf, NOC and numDesc metrics showed no quality gain, and the numCls metric decreased in quality. Amongst the other metrics in the TD function, Dep_In showed a decrease in quality for the JFlex program and avrgFieldVisibility showed a decrease for the Mango program. The quality gain values for the coupling functions were smaller in comparison to the TD function, although they were more consistent across the metrics. Although this function only contained 6 metrics, 4 out of the 6 contained significant improvements (a larger proportion compared to the TD fitness function).

The Dep_Out and Dep_In metrics were amongst the most improved (which was to be expected due to them containing aspects of the other coupling metrics used), although the parameter metrics (iC_Par and eC_Par) were also influential. The eC_Par metric showed the largest overall quality gain of the coupling metrics, improving more than even the Dep_In and Dep_Out metrics. Of the 6 metrics, the attribute metrics (iC_Attr and eC_Attr) were affected the least, although none of the metrics showed an average decrease in quality (where the average represents the mean across 10 runs of each task) across any of the benchmarks as some TD metrics did. The inheritance function showed no improvement with any of the metrics used across any of the benchmark programs. The abstraction function, while only using 3 metrics, showed quality improvements with just one of those metrics. The abstractness metric showed a small increase in quality whereas the iFImpl and numInterf metrics showed no change across any of the programs tested. The iFImpl metric similarly showed no change when used in the inheritance function and was the smallest of the improved metrics in the TD function. numIterf showed no change in the TD function either. The changes shown by the individual metrics may provide a good basis to influence how the weights should be distributed among the fitness functions. The values shown in figures 7 and 8 are not affected by metric weights (this is applied when the metrics are combined to derive the overall metric score).

5. Threats to Validity

5.1 Internal Validity

Internal validity focuses on the causal effect of the independent variables on the dependant variables. The stochastic nature of the search techniques can prove a threat to the validity of the experiment, as each run will provide different results. This has been addressed by running each of the tasks 10 times and using average values to compare against each other. The choice of parameter settings used by the search techniques can also provide a threat to validity due to the option of using poor input settings. This has been addressed by using input parameters deemed to be most effective from previous studies in the research area.

5.2 External Validity

External validity is concerned with how well the results and conclusions can be generalized. In this study, the experiment was performed on 6 different real world open-source systems belonging to different domains and with different sizes and complexities. However the experiment and the capabilities of the refactoring tool used are restricted to Java programs, therefore we cannot assert that our results can be generalized to other applications or to other programming languages. Indeed, the results showed different degrees of

variation between the metric functions with different source programs. For example, Mango and Beaver, a collections library and parser generator respectively, showed higher quality gains with coupling than with TD. Conversely, JSON and Apache-XmlRpc (a library for data exchange format and a library for remote procedure calls) showed the greatest TD improvements compared against coupling. Although the sample systems studied are very different, further replications of this study are necessary to confirm the generalization of the findings.

5.3 Construct Validity

Construct validity refers to how well the concepts and measurements are related to the experimental design. The validity of the experiment is limited by the fitness functions used, as they are experimental approximations of the properties defined based on previous research. Furthermore, the metrics used to construct the fitness functions for this study may not particularly indicate an improvement in the software, and this warrants further investigation. In order to address this threat, justifications for the choice of metrics have been discussed along with a description of the construction of the fitness functions. Ideally in future work we would synthesize the choice of inputs and weightings used for our fitness function from expert opinion or based on empirical research. The cost measures used in the experiment can also indicate a lack of validity. To assess the effectiveness of the 3 search techniques, execution time was used to measure and compare the cost.

5.4 Conclusion Validity

Conclusion validity looks at the degree to which a conclusion can reasonably be drawn from the results. A lack of a meaningful comparative baseline can provide a threat by making it harder to produce a conclusion from the results without the relevant context. To address this, a random search and a local hill climbing search are used to compare against the metaheuristic simulated annealing search, with the random search providing the baseline to compare against. Furthermore, in order to provide descriptive statistics of the results, tasks have been repeated and mean values have been used to compare against. Error bars have also been provided in most cases to indicate the range of values. Another possible threat may be provided by the lack of a formal hypothesis in the experiment. At the outset, 2 research questions have been provided along with corresponding hypotheses in order to aid in drawing a conclusion. To accompany these, statistical tests have been used to test the significance of the results gained. These tests make no assumption that the data is normally distributed and are suitable for ordinal data. As the TD function is experimental and only indicates a possible representation of the property of Technical Debt, other metric combinations may give different experimental results. A different TD function may provide better or worse calculations if the experiment was repeated and in this case, may draw a different conclusion.

6. Related Work

Search based software engineering was introduced as a term in 2001 (Harman and Jones, 2001). Further research in the area was identified, as well as open problems in 2007 (Harman, 2007). A recent review of research work in software engineering (Kumari et al., 2014) includes a short review of the area of search based software engineering and some of the work published in that area. A review of search based software engineering papers in Brazil gives useful statistics of the impact of researchers from the country on the area (Colanzi et al., 2013). More comprehensive but less recent literature reviews of search based software engineering provide a useful background to the area (Harman et al., 2012a), (Harman et al., 2012b), (Räihä, 2009). More specific literature reviews addresses the impact of the areas of project management (Ferrucci et al., 2014) and testing (Harman and McMinn, 2010), (McMinn, 2004).

There has been little research done to investigate TD specifically. A review of the impact of TD on software systems as well as methods to handle it and the cost from different perspectives is given in an article by Allman (2012). The properties of TD have also been discussed elsewhere (Brown et al., 2010), where a particular connection has been noted between TD and maintenance activities. Developers at Google have given their experience of attempts to pay off TD in the form of build debt (Morgenthaler et al., 2012). They use various attempts to uncover and remove the debt in Google code, which consists of millions of lines of code much of which is monolithic.

6.1 Refactoring Tools

Various automated refactoring tools have been proposed and used for research in SBSM. Many of these tools are used to seek out and refactor “design smells” in the code (Fowler, 2002). The TrueRefactor tool (Griffith et al., 2011) uses this method to detect and remove instances of large classes, lazy classes, long methods, temporary fields or instances of shotgun surgery. The Wrangler tool (Li and Thompson, 2010) is used to improve the modularity of programs by removing code smells. Instead of using search based tactics to

find defects, the tool inspects a module graph and a function call graph that it generates for the program. Evolution Doctor (Di Penta, 2005) is another defect removal tool that handles clones and unused objects, removes circular dependencies and reorganises source code files. Kirk et al. (2007) presents a code smell detection tool that can be used as a plug in for the Java IDE Eclipse. The tool is used to detect god classes and data classes, but cannot be used to resolve them. Trifu et al. (2004) has created the Advanced Refactoring Wizard by combining three pieces of software together to handle each stage of the approach. They use "correction strategies" to detect problems in the code, analyse them and then to refactor them. While this approach to maintaining software by finding design smells is useful there can be some restrictions. Many of these tools can address only a limited number of defects and not all can resolve the defects that are uncovered.

The tool used in this paper, A-CMA (Koc et al., 2012), uses metaheuristic search techniques to measure the code quality and to search for a better solution. It has the advantage of numerous metrics and refactoring abilities to aid with its purpose. Another tool that uses this approach by proactively improving the code instead of working to decrease issues is Code-Imp (Moghadam and Ó Cinnéide, 2011). Like A-CMA, Code-Imp provides a selection of refactorings and a number of software metrics for use. Both tools are used to refactor Java code, although C, C++, COBOL and Erlang are supported by other tools. The FermaT tool (Fatiregun et al., 2004) can use hill search or a genetic algorithm to refactor code and contains a selection of 20 refactorings available to use.

6.2 Metrics

In 2004 Harman and Clark (2004) proposed that metrics should be used as fitness functions. In the same year, Vivanco and Pizzi (2004) compared 64 different software metrics using a parallel genetic algorithm. Among the top encoded genes are method name length metrics, coupling metrics and complexity metrics. Bakar et al. (2012) has attempted to develop a metric model for selecting the most suitable metrics to measure maintainability. They analyse the CK metrics suite (Chidamber and Kemerer, 1994). Likewise, O'Keeffe and Ó Cinnéide (2006) use the QMOOD metrics suite (Bansiya and Davis, 2002) to measure the software behaviours of flexibility, reusability and understandability in terms of metrics and compare the effectiveness of each. They also observe the effects of each individual metric in the refactoring process across the 3 metric functions. Of the 3 functions, reusability is found to be unsuitable due to the introduction of a large number of featureless classes, although evidence is provided in favour of the flexibility function and in strong favour of the use of the understandability function.

Five different cohesion metrics are compared by Ó Cinnéide et al. (2012) across 8 different real world Java programs to measure their volatility. It is found that they disagreed in 55% of the applied refactorings, and in 38% of the cases metrics are in direct conflict with each other. Two of the metrics are then studied in more detail to determine where the contradictions that cause conflicts occur in the code. This is expanded on by Veerappa and Harrison (2013) by comparing 4 de-facto standard coupling metrics across 8 Java projects. They find that coupling metrics are less likely to conflict with each other, with only 7% of the changes directly conflicting with each other, but with a 55% chance that changes in one metric will have no effect on another. They also observe that improving coupling does not always directly improve cohesion. Different design principles of object oriented engineering have been proposed by Martin (2000), as well as a number of design patterns. He discusses symptoms of TD and introduces the SOLID principles used to improve the architecture of object oriented systems. No previous work is known by the authors to attempt to create a metric function to tackle TD.

7. Conclusions

In this paper we have conducted an experiment to compare 4 different fitness functions with selected weights and metrics. Three functions were chosen to represent common properties of an object oriented program and a fourth, novel function was chosen to represent the TD in the program. Previous work (O'Keeffe and Ó Cinnéide, 2006) has compared different metric functions before, although to the authors knowledge, there has been no known attempt to create a fitness function representing the TD of a software system. Three different searches were used: random search, hill climbing and simulated annealing, with simulated annealing yielding the more significant results and the hill climbing search failing to better the random search. Six open source Java programs were used as a basis for the refactoring process and the quality gain for each was compared using the 4 fitness functions. The results generated with the simulated annealing search were analysed. Of the 3 behavioural functions, only coupling was found to be useful with the other 2, abstraction and inheritance, showing little to no improvement in the results. Related literature (Veerappa and Harrison, 2013), (Vivanco and Pizzi, 2004) tends to suggest that cohesion and coupling metrics are more suitable for refactoring and the results may support

that coupling is a good behaviour to measure for improved quality. Further inspection also showed that while simulated annealing allows negative movements throughout the initial stages of the search, the amount of refactoring actions applied for the 2 weaker functions mirrored their metric results. Thus it was speculated that the metrics used to compose those functions were not as volatile as the ones used in the other two.

Generally, the TD function proved to generate a larger proportional improvement in the Java programs, although in 2 of the programs, the average coupling value was better than the score given by the TD function. It is possible that these improved results were due to the properties of the programs in question and those programs were more coupled and thus had more opportunity for improvement. Furthermore, as the TD function is made up of significantly more metrics than the other 3, perhaps this allows more freedom for the search during the refactoring process and has a positive effect on the results gained with the added options available. The amount of applied refactoring actions for each fitness function were compared with the quality gain results, a fresh insight used to gain more understanding of the effect the different functions can have. This comparison supported the idea that different metrics may provide more available refactorings, thus corresponding to a larger number of positive changes. The influence of the individual metrics on the fitness functions were explored, with the results likewise indicating that some metrics were more influential than others and more importantly, that some were not influential at all.

To address the research questions proposed at the outset of the paper, the Wilcoxon signed-rank test was used to measure how significant the differences were between the TD results (the Wilcoxon signed-rank test is used to compare non-parametric paired data sets) and the Wilcoxon rank-sum test was used to measure how significant the differences were between the search results (the Wilcoxon rank-sum test compares unpaired data sets). The quality gain given by the TD function using simulated annealing was calculated to be significant, rejecting the null hypothesis that there would be no significant effect on TD after refactoring. To test the null hypothesis of the second research question, the quality gain values of the TD function were compared across the 3 search techniques. Simulated annealing was found to be significantly different to the others, and the random search was found to be significantly different to the hill climbing search. The significance of the simulated annealing results rejects the null hypothesis of research question 2.

7.1 Actionable Conclusions

The actionable findings to be taken from these results are as follows. The simulated annealing optimisation performs better than a local hill climbing search or a random search in the task of removing TD in a Java program through automated refactoring. Using a fitness function to represent the TD of a program, the programs have been show to give a better improvement in metric values compared against fitness functions that only aim to measure specific properties of the software (coupling, inheritance, abstraction). The results show that it may indeed be possible for software developers to use a fully automated approach to decrease the TD in a software system, and that it may be useful to combine metrics to represent more abstract properties of a system. This automated approach could ease the costly maintenance process usually involved in software development, saving time and effort for the developer.

8. Future Work

Various avenues have been uncovered for further research. It has been found that, as the program size increases and the available search size increases, the time taken to execute the tasks will increase at a non linear rate (this may be because the amount of available refactorings will increase at order n^2). Further research using A-CMA's available parallel functionality and exploring other options would hopefully allow for a more agreeable execution time on larger programs. Another option is to explore alternate or more recently developed search techniques such as 'Great Deluge' or a global search. Alternatively, a multi-objective approach may provide improved results in the metric functions. More work required includes further inspection of the effect that individual metrics can have on the refactoring ability of an automated refactoring program (to deduce the volatility of each metric) and inspecting whether more metrics can yield more useful results. Further research of the sensitivity of individual metrics on the fitness functions could help derive the best combination of metrics and weights to use for an effective fitness function. Otherwise, a consensus derived among software experts may inform a more reliable set of weights to use for the individual metrics in the function. A correlation was found in this study between the amount of refactoring actions available to a fitness function and the improvement of the fitness function after refactoring. They may be merit in conducting further investigation to test this connection for any valid implications. Further inspection of the hill climbing search would also be useful to inspect why it failed to produce better results than the baseline random search. Finally, further investigation could be considered by

comparing TD against other measures of software quality beyond abstraction, coupling and inheritance such as cohesion, encapsulation, polymorphism or complexity.

Acknowledgements

The authors are deeply grateful to Ekin Koc for permission to modify and use his refactoring tool A-CMA. The research for this paper contributes to a PhD project funded by the UK EPSRC.

References

Allman, E., 2012. Managing Technical Debt. Queue. 10, 50–55. doi:10.1145/2168796.2168798

Bakar, A.D., Sultan, A.B., Zulzalil, H., Din, J., 2012. Applying Evolution Programming Search Based Software Engineering (SBSE) In Selecting The Best Open Source Software Maintainability Metrics, in: International Symposium on Computer Applications and Industrial Electronics, ISCAIE 2012. Ieee, pp. 70–73. doi:10.1109/ISCAIE.2012.6482071

Bansiya, J., Davis, C.G., 2002. A Hierarchical Model For Object-Oriented Design Quality Assessment. IEEE Trans. Softw. Eng. 28, 4–17. doi:10.1109/32.979986

Basili, V.R., Caldiera, G., Rombach, H.D., 1994. The Goal Question Metric Approach, in: Encyclopedia of Software Engineering. John Wiley & Sons, pp. 528–532.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., 2010. Managing Technical Debt In Software-Reliant Systems, in: FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010. pp. 47–52. doi:10.1145/1882362.1882373

Chidamber, S.R., Kemerer, C.F., 1994. A Metrics Suite For Object Oriented Design. IEEE Trans. Softw. Eng. 20, 476–493.

Colanzi, T.E., Vergilio, S.R., Assunção, W.K.G., Pozo, A., 2013. Search Based Software Engineering: Review And Analysis Of The Field In Brazil. J. Syst. Software. 86, 970–984. doi:10.1016/j.jss.2012.07.041

Di Penta, M., 2005. Evolution Doctor: A Framework To Control Software System Evolution, in: 9th European Conference on Software Maintenance and Reengineering, CSMR 2005. Ieee, pp. 280–283. doi:10.1109/CSMR.2005.29

Fatiregun, D., Harman, M., Hierons, R.M., 2004. Evolving Transformation Sequences Using Genetic Algorithms, in: 4th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2004. IEEE Comput. Soc, pp. 65–74. doi:10.1109/SCAM.2004.11

Ferrucci, F., Harman, M., Sarro, F., 2014. Search-Based Software Project Management, in: Software Project Management in a Changing World. pp. 1–28.

Fowler, M., 2015. Refactoring Catalog [WWW Document]. URL http://refactoring.com/catalog/ (accessed 4.22.15).

Fowler, M., 2002. Refactoring: Improving The Design Of Existing Code.

Griffith, I., Wahl, S., Izurieta, C., 2011. TrueRefactor: An Automated Refactoring Tool To Improve Legacy System And Application Comprehensibility, in: 24th International Conference on Computer Applications in Industry and Engineering, ISCA 2011.

Harman, M., 2007. The Current State And Future Of Search Based Software Engineering, in: Future Of Software Engineering, FOSE 2007. pp. 342–357.

Harman, M., Clark, J., 2004. Metrics Are Fitness Functions Too, in: 10th International Symposium on Software Metrics, METRICS 2004. Ieee, pp. 1–12. doi:10.1109/METRIC.2004.1357891

Harman, M., Jones, B.F., 2001. Search-Based Software Engineering. Inf. Softw. Technol. 43, 833–839. doi:10.1016/S0950-5849(01)00189-6

Harman, M., Mansouri, S.A., Zhang, Y., 2012a. Search Based Software Engineering: Trends, Techniques And Applications. ACM Comput. Surv. 45, 1–64. doi:10.1145/0000000.0000000

Harman, M., McMinn, P., 2010. A Theoretical And Empirical Study Of Search-Based Testing: Local, Global, And Hybrid Search. IEEE Trans. Softw. Eng. 36, 226–247. doi:10.1109/TSE.2009.71

Harman, M., McMinn, P., De Souza, J.T., Yoo, S., 2012b. Search Based Software Engineering: Techniques, Taxonomy, Tutorial, in: Empirical Software Engineering and Verification. pp. 1–59.

Kirk, D., Roper, M., Wood, M., 2007. A Heuristic-Based Approach To Code-Smell Detection, in: 1st Workshop On Refactoring Tools, WRT 2007. pp. 54–55.

Koc, E., Ersoy, N., Andac, A., Camlidere, Z.S., Cereci, I., Kilic, H., 2012. An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques, in: Gelenbe, E., Lent, R., Sakellari, G. (Eds.), Computer and Information Sciences II. Springer London,

London, pp. 59–66. doi:10.1007/978-1-4471-2155-8
Kumari, M., Sharma, M., Kumar, A., 2014. A Review Of Research Work In Software Engineering. Int. J. Eng. Comput. Sci. 3, 5288–5298.
Li, H., Thompson, S., 2010. Refactoring Support For Modularity Maintenance In Erlang, in: 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2010. Ieee, pp. 157–166. doi:10.1109/SCAM.2010.17
Martin, R.C., 2000. Design Principles And Design Patterns. Object Mentor.
McMinn, P., 2004. Search-Based Software Test Data Generation: A Survey. Softw. Testing, Verif. Reliab. 14, 1–58.
Mitchell, B.S., Mancoridis, S., 2002. Using Heuristic Search Techniques To Extract Design Abstractions From Source Code, in: Genetic and Evolutionary Computation Conference, GECCO 2002. pp. 1375–1382.
Moghadam, I.H., Ó Cinnéide, M., 2011. Code-Imp: A Tool For Automated Search-Based Refactoring, in: 4th Workshop on Refactoring Tools, WRT 2011. pp. 41–44.
Morgenthaler, J.D., Gridnev, M., Sauciuc, R., Bhansali, S., 2012. Searching For Build Debt: Experiences Managing Technical Debt At Google, in: 3rd International Workshop on Managing Technical Debt, MTD 2012. Ieee. doi:10.1109/MTD.2012.6225994
Murgia, A., Tonelli, R., Concas, G., Marchesi, M., Counsell, S., 2012. Parameter-Based Refactoring And The Relationship With Fan-In/Fan-Out Coupling. J. Object Technol. 11, 1–24. doi:10.1109/ICSTW.2011.26
Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., Moghadam, I.H., 2012. Experimental Assessment Of Software Metrics Using Automated Refactoring, in: ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2012. pp. 49–58.
O'Keeffe, M., Ó Cinnéide, M., 2008. Search-Based Refactoring For Software Maintenance. J. Syst. Software. 81, 502–516. doi:10.1016/j.jss.2007.06.003
O'Keeffe, M., Ó Cinnéide, M., 2007. Getting The Most From Search-Based Refactoring, in: 9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007. pp. 1114–1120.
O'Keeffe, M., Ó Cinnéide, M., 2006. Search-Based Software Maintenance, in: 10th European Conference on Software Maintenance and Reengineering, CSMR 2006. pp. 251–260.
O'Keeffe, M., Ó Cinnéide, M., 2003. A Stochastic Approach To Automated Design Improvement, in: 2nd International Conference on Principles and Practice of Programming in Java, PPPJ 2003. pp. 59–62.
Räihä, O., 2009. An Updated Survey On Search-Based Software Design.
Trifu, A., Seng, O., Genssler, T., 2004. Automated Design Flaw Correction In Object-Oriented Systems, in: 8th European Conference on Software Maintenance and Reengineering, CSMR 2004. Ieee, pp. 174–183. doi:10.1109/CSMR.2004.1281418
Veerappa, V., Harrison, R., 2013. An Empirical Validation Of Coupling Metrics Using Automated Refactoring, in: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. Ieee, pp. 271–274. doi:10.1109/ESEM.2013.37
Vivanco, R., Pizzi, N., 2004. Finding Effective Software Metrics To Classify Maintainability Using A Parallel Genetic Algorithm, in: 6th Annual Conference on Genetic and Evolutionary Computation, GECCO 2004. pp. 1388–1399.

Michael Mohan received his MEng degree in Computer Games Design And Development at Queen's University Belfast. During the degree, he undertook a 12 month placement in industry as a software engineer. He is currently a Ph.D. student at Queen's University. His research interests include search based software engineering, software maintenance, automated refactoring and multi-objective search techniques.

Dr. Desmond Greer is a senior lecturer at Queens University, Belfast. He earned a Master of Science and Doctorate at Queens University and University of Ulster. His research can be collectively described as how to better manage and adapt to change, both in the software product and the software development process. He has published over 50 research papers, many of which have arisen from research in an experimental context, in collaboration with industry. He is a member of the IEEE and IEEE Computer Society as well as an active member of the International Software Engineering Research Network (ISERN).

Dr. Paul McMullan is a lecturer in the School of Electronics, Electrical Engineering and Computer Sciences (EEECS) at Queen's University Belfast. His main research expertise is in the area of Scheduling and Optimization using Artificial Intelligence and Hyper-Heuristic Search Techniques. He is also a practitioner of applying research to Industrial Applications and Commercial Spin-outs. He is a Director of two Queen's University IT companies, Realtime Solutions Limited and EventMap Limited.